NASA Technical Memorandum 104347

ALPS—A Linear Program Solver

Donald C. Ferencz
Case Western Reserve University
Cleveland, Ohio

and

Larry A. Viterna
Lewis Research Center
Cleveland, Ohio

April 1991

NASA

ALPS - A LINEAR PROGRAM SOLVER

Donald C. Ferencz*
Case Western Reserve University
Cleveland, Ohio 44106

Larry A. Viterna
National Aeronautics and Space Administration
Lewis Research Center
Cleveland, Ohio 44135

Summary

ALPS is a computer program which can be used to solve general linear program (optimization) problems. ALPS was designed for those who have minimal linear programming (LP) knowledge and features a menu–driven scheme to guide the user through the process of creating and solving LP formulations. Once created, the problems can be edited and stored in standard DOS ASCII files, to provide portability to various word processors or even other linear programming packages. Unlike many math–oriented LP solvers, ALPS contains an LP "parser" that will read through the LP formulation and report several types of errors to the user. ALPS provides a large amount of solution data which is often useful in problem solving.

In addition to pure linear programs, ALPS can solve for integer, mixed integer, and binary type problems. Pure linear programs are solved with the revised simplex method. Integer or mixed integer programs are solved initially with the revised simplex, and then completed using the branch–and–bound technique. Binary programs are solved with the method of implicit enumeration.

This manual describes how to use ALPS to create, edit, and solve linear programming problems. Instructions for installing ALPS on a PC compatible computer are included in the appendices along with a general introduction to linear programming. A programmers guide is also included for assistance in modifying and maintaining the program.

*Summer Student Intern at NASA Lewis Research Center.

Contents

Nomenclature

ALPS	A Linear Program Solver (computer program)
APL	A Programming Language
IBM	International Business Machines Corporation
LP	linear program
MAX	maximize
MIN	minimize
S.T.	such that
!	designates an input comment

1.0. Introduction

ALPS is, as the name describes, a linear program (LP) solver. This manual describes how to use ALPS to create, edit, and solve LP problems.

ALPS features a menu–driven scheme to guide the user through the process of creating and solving LP formulations. Once created, the problems can be edited and stored in standard DOS ASCII files, to provide portability to various word processors or even other linear programming packages. Unlike many math–oriented LP solvers, ALPS contains an LP "parser" that will read through the LP formulation and report several types of errors to the user. ALPS provides a large amount solution data that is often useful in problem solving.

In addition to standard LP's, ALPS can solve for integer, mixed integer and binary type problems. Pure linear programs are solved with the revised simplex method. Integer or mixed integer programs are solved initially with the revised simplex, and then completed using the branch–and–bound technique. Binary programs are solved with the method of implicit enumeration.

ALPS was written for a PC compatible computer. Appendix A contains information on system requirements and installation instructions.

In general, ALPS was designed for those who have a minimum of knowledge in linear programming. However, some familiarity with the "jargon" of optimization is assumed. Appendix B contains a brief introduction to linear programming with several example problems.

A programmers guide is included in Appendix C for assistance in modifying and maintaining the program.

2.0. Problem Editing

There are two ways to enter a problem formulation into ALPS: through the built–in editor of ALPS, or, by retrieving a formulation stored as an ASCII file. This section describes the operation of the ALPS editor.

2.1. Invoking the Editor

Choosing option (E) Enter/Edit Formulation from the main menu will invoke the ALPS problem editor. The text window covering most of the screen is used much like any other editor to create a problem formulation. The cursor keys may be used to move around the current edit window, and the LP equations/commands can be placed anywhere in the window (subject to the rules of formulation, as described later).

2.2. Editor Key Functions

Note that the edit "window" covers only a portion of the entire problem formulation. The text may be scrolled up, down, left, and right to display other parts of the problem. Within the editor, the following keys have special functions:

Key	Editor Function
Cursor Keys	The cursor keys function normally
Home	Move the cursor to the beginning of the line.
End	Move the cursor to the end of the line.
Page Up	Scroll up by one editing window (but not past the top of the problem
Page Down	Scroll down by one editing window (but not past the bottom of the
problem).	
Ctrl–Page Up	Scroll up by one line.
Ctrl–Page Down	Scroll down by one line.
Ctrl–Left	Scroll the text window left by two columns.
Ctrl–Right	Scroll the text window right by two columns.
F1	Display the editor's built–in help screen
F2	Check the problem formulation and exit if the problem is correct.
Otherwise,	
	a pop–up window will describe the error (see "Formulation Errors" for more information).
F3	Abort the current formulation; that is, exit without saving changes. Any
changes	
	made during the current edit WILL BE LOST.
F8	Insert a blank line after the cursor.
Shift–F8	Delete the entire line that the cursor is positioned on.
Ctrl–End	Delete all characters from the current cursor position to the end of the line.

3.0. Problem Formulation

In order to correctly solve any linear program, ALPS requires that the formulation follow certain rules. In general, the linear program must be expressed in "standard form." Other restrictions apply, and are listed here.

3.1. Decision Variables

The decision variables used in the linear program must all begin with an alphabetic character, and may contain any combination of alphanumeric characters and the underscore ("_"). ALPS only reads the first six characters of each variable name; be careful, since "AIRFLOW_1" is actually the same as "AIRFLOW_2". Decision variables do not have to be "declared"; they are implicit within the problem. Variables are case–sensitive (e.g., "Ab" is a different variable than "aB").

Special rules apply to variables beginning with "B" and "I" (upper case only). Any name beginning with "I" is used to denote a variable that can only have an integer value in the solution (for example, a variable representing the number of ships entering a port). Likewise, any variable beginning with "B" represents a binary variable. A binary variable is a special type of integer variable that can only take on the values "0" or "1". These variables have special uses in linear programming. See Appendix B for more information on binary/integer programming.

The following are valid decision variable names in ALPS: X1, FLOW_1, Xj23Q2, A___

The following are NOT valid decision variable names: 1x2, _apples, %Y7, AND*or

3.2. Objective Function

ALPS requires that the problem formulation begin with the objective function. Other than comment lines beginning with "!", the first line of the LP must begin with "MAXIMIZE" or "MINIMIZE" (or "MAX" and "MIN", for short). Following this is the sum of the decision variables, preceded by their coefficients (costs). The objective function may cover several lines.

At least one, but not all, decision variables must appear in the objective function. ALPS will assume that any decision variable not appearing in the objective has a zero cost. However, variables may NOT appear in the objective function if they do not appear in the constraint set, as this would lead an "unrestricted" variable.

The following is an example of a valid objective function:

MAX X1 + 2X2 – X3 + 12X4

Note that ALPS does not allow constants in the objective function equation, as they will not affect the solution.

3.3. Constraint Set

Following the objective function is the constraint set. The set of constraints for the linear program must begin with the statement "SUCH THAT" (or abbreviated "S.T.") on a separate line. ALPS allows constraints to appear on multiple lines, but only one constraint (or the beginning of one constraint) must appear on any one line.

The constraints must be expressed in the "standard" form of linear programming; that is, all variables must be on the left–hand side of the equation, with only a constant appearing on the right hand side. Thus the inequality

X1 + X2 + X2 < Y1 + Y2 must be expressed within ALPS as

X1 + X2 + X3 – Y1 – Y2 < 0.

In linear programming, only "non–strict" inequalities ("≤" and "≥") are actually valid. ALPS allows input of both strict ("<" and ">") and non–strict inequalities; but in all cases the non–strict inequalities are assumed. Of course, equality constraints are also valid. Standard form of linear programs also requires that all right–hand side constants be positive. If any right–hand side is negative, however, ALPS will automatically multiply through by –1 internally to convert the equation to standard form.

Each constraint may be given a "name." This name is entered the same as a comment and is presented in the solution output. To name a constraint, place a comment character "!" after the right–hand side expression, followed by the constraint name:

COST1 + COST2 + COST3 ≤ 100 ! Fiscal_Budget

3.4. Miscellaneous

Numerical constants and coefficients must be expressed in simple decimal form (e.g., "12.329"). Exponential notation is not permitted. If a problem contains large numbers, it should be rescaled (e.g., use "millions of gallons" instead of "gallons") to maintain significant accuracy in the problem.

3.5. Formulation Errors

Within the editor, the formulation can be checked with the 'F2' key. Alternately, if the formulation is loaded from a disk file, ALPS will automatically perform syntax/error

checking. If there is a formulation error, ALPS will present one of the following error messages, along with the number of the line where the error was found. Listed below are the possible errors, reasons for the error, and suggested solutions.

Illegal Character(s): ALPS has detected a character that has no place in the LP formulation (such as "*" or "$") and is not within a comment. Check for typos or invalid decision variable names.

Missing or Bad Variable Name: A variable was expected, but none was found (such as "X + < 7"). This may also be caused by a variable name beginning with a numeric digit. Change the variable name.

Missing Constraint Sign: A constraint was included that has no (valid) constraint sign ("<", ">", "=", "≥", or "≤"). Check to make sure all constraints are valid.

Illegal Right–Hand Side: The right–hand side of a constraint was bad; either it contains a variable ("X < Y") or an ill–formed constant ("X < 3."). Make sure all right–hand sides are pure constants.

Invalid Coefficient: The coefficient of a decision variable was found to be invalid. Check for typos.

No Constraints!: The current linear program appears to have no constraints. Check to make sure the keywords "SUCH THAT" appear before the constraint set.

Error in Objective Function: Some syntax error was found in the objective function, such as a misspelling of "MAX", etc. Check the entire objective function carefully.

Bad Variable in Objective Function: A variable was found in the objective function that does not appear in the constraint set. This "unrestricted" variable would cause an unbounded solution. Check for misspellings in the objective function; make sure the problem formulation is correct (i.e., has enough constraints).

Variable Appears Twice: ALPS does not allow a variable to be used twice within one constraint. Normally, this means there is a typo in the offending constraint. Otherwise, "condense" the two coefficients into a single one by adding them.

Sometimes, particularly in the case of constraints that cover multiple lines, ALPS will report a line number that is "close–but–not–quite" where the error is located. Check around the area where ALPS reports the error and it will most likely be found.

4.0 Solution

Once a problem formulation has correctly been entered into ALPS, the solution process may begin. Choosing (S) Solution from the main menu brings up the solution menu.

4.1. Solving the Problem

Choosing (S) Solve Problem from the solution menu will begin the solution process. Although there are several solution procedures for the several types of linear programs that ALPS will solve, they are chosen automatically. A status line in the middle of the screen describes which solution technique is taking place:

Solving Linear Program: In this case, the problem is purely a linear program and ALPS has only to perform the "simplex method" to solve it. ALPS display the current iteration count, as well as the current "entering" and "leaving" variable in the simplex basis. These are useful for determining whether the LP is "cycling" — see the section on "Errors in the LP Solution" for more information.

Solving Initial LP: This is the first phase of solving an integer program, and the display is the same as noted above. Once the "initial" LP is solved, ALPS will present the display for the "branch–and–bound" procedure, described below.

Performing Branch and Bound: ALPS is solving the second phase of an integer program. Generally, this takes quite a bit of time. The status displays the "depth" of the solution as well as the number of "boxes" generated; this is useful for tracking the solution process.

Performing Implicit Enumeration: If the problem consists purely of 0–1 integer (binary) variables, ALPS uses a much faster and more efficient technique to solve this special type of problem. The status will generally change very quickly.

If one or more solutions were found, ALPS will display a positive status (Unique Optimum Found or Multiple Optima Found); otherwise, an error status will be displayed.

4.2. Displaying the Solution

Once the LP has been solved, there is a multitude of information to examine. From the solution menu, choosing (D) Display Solution will present the main solution data. The top of this screen displays the heading Solution X of Y; if more than one solution was found, the keys F1 and F2 can be used to cycle through each solution in turn.

The output of the solution is displayed in a "scrolling output screen." The information can be paged up and down with the Page Up and Page Down keys or line–by–line using the up and down arrow keys.

Each solution has the heading LP Solved with Optimal Value of XXX, denoting the final value (maximum or minimum, depending on the problem formulation) that was found for the solution. Below this appears the optimal solution: Variable Name, Optimal Value, and Reduced Costs for each variable in the formulation. For more information on Reduced Costs see Appendix B.

ALPS also lists information about each row in the constraint set. Note that the Constraint Row is not the absolute row number of the problem formulation, but the ordinal number of the constraint within the constraint set. If the constraint has a Name, this is displayed also. If the constraint was an inequality, the Slack or Surplus (the amount the left–hand side differs from the right–hand side, for each a "≤" or "≥" constraint, respectively) is shown. Finally, the Dual Cost (sometimes known as the "shadow cost") for each constraint is displayed. (More information on "dual costs" is available in Appendix B, as well as any linear programming text that considers "duality.")

4.3. Range Analysis

ALPS also produces a list of "optimal ranges" for each of the objective function coefficients and the right–hand sides of the constraints. These are the ranges in which the coefficients or right–hand side values can change without affecting the final solution (that is, the optimal value of the decision variables). Choose (R) Range Analysis from the solution menu to view all of these ranges. Like the solution display, the data is presented in a scrolling viewer. Alternate ranges (related to any alternate solutions that have been viewed above) can be toggled with F1/F2.

Please note that ranges are not available for the pure binary program; the solution technique does not provide this information. Also, the range information for any mixed–integer or integer program (with any "I" variables) should be taken lightly; this type of solution is not completely determined by the objective function coefficients alone. For strictly linear programs (with no "I" or "B" variables), however, the range analysis (known as "parametric" or "sensitivity" analysis) can be a very powerful tool for resolving a linear program without having to rerun ALPS.

4.4. Solution Report

A complete solution report, with optimal values, shadow costs, slacks and surpluses, and optimal ranges can be created by choosing (P) Print Solution Report from the solution menu. This report can be directed to any DOS file, including some sort of list device (such as "PRN" or "LPT1"). The report contains a complete listing of the problem formulation, as well as the information described in each of the above sections, for each optimal solution.

5.0. File Operations

As mentioned before, ALPS has facilities for storing and retrieving linear programs as ASCII files. Choosing (F) File Functions from the main menu will display the ALPS file screen.

5.1. Saving Files

Any current ALPS formulation can be saved to a DOS file assuming the formulation has been checked for proper syntax. Selection (W) Write File will prompt for a DOS filename. Simply enter a name and ALPS will store the text of the problem formulation in standard ASCII format. This is useful for printing the formulation, or even incorporating the problem into a word processor.

Note that ALPS does not store any solution information along with the problem, so that if the file is re–loaded into ALPS it will have to be re–solved.

5.2. Reading Files

ALPS can read a previously stored file with the (R) Read File command. Enter the filename when prompted, as above. This file can be one stored by ALPS in an earlier session, or an ASCII file created with a word processor. DO NOT try to read a formatted word processor file; the control codes will most likely confuse ALPS into a crash. Instead, make sure the word processor has the capability to store files in plain ASCII format.

Once the file has been loaded, ALPS will immediately check it for proper syntax. For large problems, this may take a while; ALPS displays "Checking Problem, Please Wait..." while this is taking place. If there are any errors in the formulation, ALPS will enter into the edit screen to allow for correction.

Appendix A: Computer Requirements and Installation

ALPS is provided on 5.25 inch. high density floppy diskettes containing the following two files:

alps.exe

alps.atf

There are two possible ways of running ALPS, depending on which version of the program is being used.

ALPS can exist as a stand–alone program from DOS. In this "packed" version of ALPS, only about 270K of memory is available.

If ALPS is contained as an IBM APL2/PC workspace (as from the original development), the workspace must be loaded from transfer format. The advantage to using ALPS from within the APL2 environment is that the program can make full use of any installed extended memory.

Stand–alone EXE Version

For this version, ALPS requires an IBM PC or compatible with at least 640 K of memory and DOS 3.1 or higher.

To install ALPS, simply create a directory and copy the alps.exe file to it.

To run ALPS, change to the directory containing the alps.exe file and type "ALPS" at the DOS prompt.

APL2/PC Workspace Version

For this version, ALPS requires the IBM APL2 programming language on a 80386 or 80486 based microcomputer with a 80387 math coprocessor, at least 2 megabytes of extended memory, and DOS 3.3 or higher. APL2 for the PC is available from:

IBM Direct
Phone: 800–IBM–2468
Part Number 6242936

To install, copy the ALPS workspace, "alps.atf", to the system APL2 directory. Next, enter the APL2 system. It should be noted that ALPS requires three auxiliary processors ap2, ap124, and ap210. An example invocation would be:

apl232 ap2 ap124 ap210

ALPS can now be imported from its transfer–format file with the command:

```
)IN ALPS
```

From here, the user may examine or modify any of the ALPS code. To begin execution of ALPS, the main function is called by typing:

```
ALPS
```

ALPS can also be stored in workspace form with the command:

```
)SAVE ALPS
```

All further use of ALPS can now use the command:

```
)LOAD ALPS
```

Appendix B: An Introduction to Linear Programming

Introduction

Linear programming represents a small part of the more general field of mathematical optimization. Optimization can be loosely defined as the practice of searching a set of alternatives for the best candidate. Often, the number of alternatives is numerous, even infinite. Optimization provides a way to find the "best" option without having to examine each and every alternative.

In optimization practice, the set of alternatives is expressed mathematically. Obviously, a linear program implies that the mathematical interpretation of the system to be optimized be expressed in a simple, linear fashion. With that in mind, the scope of the linear program is quite clear. (Many methods of representing and solving general, nonlinear problems exist, but are not considered here).

The term "programming" suggests a method involved with this linear model; indeed, the technique of linear programming gives specific steps to translate a system (physical or otherwise) into a mathematical optimization model that can then be solved. Although linear programming may also consider the actual techniques used to solve the problem, this discussion will only briefly cover steps used to find a solution, as well as the interpretation of that solution.

The Components of a Linear Program

A linear program is made up of three basic parts, all of which must be clearly identified (or chosen) before the model can be composed: the decision variables, constraint set, and objective function.

Decision Variables

The ***decision variables*** of a linear program imply a "choice" (or decision) to be made within the solution of the problem. Each decision variable can be thought of as a "degree of freedom" of the solution. Decision variables can represent anything, from physical dimensions to prices to shipping quantities. Keep in mind, though, that often a judicious choice of decision variables will make the resulting linear program smaller (and thus more efficient) than another choice.

In most cases the choice of decision variables lends itself well to the problem. However, to ensure that the rest of the problem can be formulated in a straightforward manner, perform a quick run–through of the rest of the problem formulation before making a final choice of decision variables.

Constraint Set

In most problems, particularly those involving physical systems, a set of ***constraints*** on the values of the decision variables is imposed. These constraints are expressed as any linear combination of the decision variables of the problem, typically utilizing both equalities and inequalities. Here, the choice of decision variables becomes critical: well–planned decision variables usually lead to straightforward constraints.

Sometimes in the course of creating constraints it is convenient to introduce a "new" (intermediate) decision variable; this is perfectly valid within linear programming, although it may tend to make the problem more complex. An example of this are the "slack" and "surplus" variables used in converting a linear program to standard form; these are described later.

With the introduction of the constraint set, a little terminology is in order. Any set of values for the decision variables that satisfies all of the constraints is known as a ***feasible point*** or ***feasible solution*** of the linear program. One can think of each decision variable as a component of a vector in *nspace* (if there are n decision variables). The ***feasible region*** of the linear program, then, is the set of all feasible solutions.

Objective Function

Solving an optimization problem requires an expression which to optimize; this is the ***objective function.*** In linear programming, one is always searching for the minimum or maximum value of the objective function, which is expressed as a linear sum of the decision variables, each with its own coefficient. The coefficient of each decision variable in the objective function is referred to as the ***cost coefficient*** or just the cost of the variable.

It is perfectly valid for the objective function to contain less than all the decision variables; those variables not appearing in the objective function have a cost of zero (and are thus "insignificant" in terms of the final solution).

With these terms defined, it is simple to see that the ***optimal solution*** to the linear program one or more feasible points that have the best (maximum or minimum) value of the objective function. The details of this solution will be described later.

An Introductory Example

It is difficult to grasp the immediate concepts of linear programming without a direct example; thus, here is a rather simple introduction to the creation of a linear programming (LP) formulation from a problem:

The production manager at a plant is overseeing the production of two chemicals, A and B. The production for each 100 gallons of these two chemicals depends on two processes (1 and 2), as follows:

Product	Time in Process 1	Time in Process 2
A	2 hr.	4 hr.
B	3 hr.	5 hr.

The total times available per week are 16 hr. for Process 1 and 24 hr. for process 2.

The manager wishes to determine the optimal schedule so as to maximize total revenues, given the following profit schedule:

Product	Profit per 100 Gallons
A	$ 40
B	$100

The solution of the problem begins with the identification of the three basic components listed above. First, the decision variables must be chosen. It is clear that the manager wishes to "decide" upon the production quantities of the two chemicals A and B. Thus we define the two principle (and only) decision variables of the problem to be:

Xa = Quantity of Chemical A to produce per week
Xb = Quantity of Chemical B to produce per week

The units for Xa and Xb should be chosen as "100 gallons/week", since the problem is stated all in terms of hundreds of gallons. In general, it is good practice to "scale" the problem in any possible way so as to maintain numerical accuracy.

With the decision variables chosen, the constraints (expressed in verbal form within the problem) can now be converted to mathematical expressions. The time in the two processes (in hours) to produce 100 gallons is given, along with a maximum limit on process hours. Thinking in terms of this limit, then, one can express the limits upon the hours used in Process 1 and Process 2 as the following inequalities:

$$2\,Xa + 3\,Xb \leq 16$$
$$4\,Xa + 5\,Xb \leq 24$$

The inequalities signify that it is perfectly valid (within this model, at least) to use less than the maximum time available. Here we are imposing a limit on process hours, but the limit is (and must be) written in terms of the decision variables.

As it happens, these are the only constraints in the original problem, so that all that remains is to write the objective function. We wish to express the profit of the production of chemicals A and B, and then maximize it. Quite simply then, the objective function for the problem is:

MAXIMIZE 40 Xa + 100 Xb

Linear Program "Standard Form"

This completes the linear program formulation. In most cases the LP should be expressed in "standard form." Simply put, this means that the objective function must be expressed first, and that all constraints be placed in the following form:

a1X1 + a2X2 + a3X3 + ... ("≤" or "≥" or "=") b

where b is the "right–hand side" of the constraint, and must be a positive quantity. "b" can be made positive if necessary by multiplying the entire constraint expression by "–1" (and reversing the sign, if it is an inequality). Notice also that all variables must be on the left–hand side of the expression.

Also part of standard form is the "nonnegativity" of all variables. This is not something that must be expressed, but something that is assumed by almost all linear program solvers (including ALPS). Every decision variable is assumed to have a positive value. Alternately, it can be viewed that the constraints "Xa ≥ 0", "Xb ≥ 0" (so on for each variable) are implicit within the problem. How does one use variables that can take on positive and negative values, then? These so–called "free" variables can be rewritten as the difference between two nonnegative variables. The procedure for this would be to write the LP as if all variables were free, and then substitute a difference of two new variables everywhere the free variable(s) occur. For example, if the variable "Xb" above were to become a free variable, one would substitute "Xb1 – Xb2" for "Xb" (within both the constraint set and the objective function — don't forget to multiply out any coefficients). When the LP is solved, the optimal value of "Xb" can be found by subtracting the optimal values of "Xb1 – Xb2". In physical systems, however, negative quantities are not often valid, and thus the nonnegativity assumptions of linear programming are useful.

Slacks and Surpluses

The straight–line definition of "standard form" for a linear program also requires that all constraints actually be of the form

$$a1x1 + a2x2 + a3x3 + \ldots = b1$$

that is, all constraints be expressed as an equality. Although ALPS (and most other linear program solvers) do not require that the LP be converted to this form, it is somewhat useful to know how this is done. The technique used is the addition of slack and surplus variables.

First, take the case of a constraint of the form "$a1X1 + a2X2 + a3X3 + \ldots \leq b$". Notice that an additional, positive value can be added to the left–hand side of the equation. If this value is the value of the difference between the sum "$a1X1 + a2X2 + a3X3 + \ldots$" and the right–hand constant "b" then the inequality becomes an equality. Thus we have converted the equation. In the case of a variable, we simply place a ***slack variable*** "Xs" on the left–hand side to form:

$$a1X1 + a2X2 + a3X3 + \ldots + Xs = b$$

Note that we require "Xs" to be positive; this is already implicit in the nonnegativity of "Xs."

Surplus variable represent the same concept as a slack variable, but for "≥" constraints: instead of adding the surplus onto the right–hand side, however, we subtract it. Thus any equation with a "≥" in it automatically becomes an equality. Again, the nonnegativity of the surplus variable is required and implied.

Once slack and surplus variables are added into the problem, they are implicitly given coefficients of zero in the objective function. This makes sense: the optimal solution of the problem should not depend upon any slack or surplus, since they have no actual bearing on the problem. As mentioned above, variables can be created at will. Although it would serve no purpose, one could create two or more slacks within a given constraint.

One point that has been subtly "glazed over" up until now is the distinction between "<" and "≤" constraints (or ">" and "≥"). Mathematically, the expression "$X < 3$" and "$X \leq 3$" are quite different. In linear programming, only constraints of the form "$X \leq 7$" are actually valid. This is due to the nature of any problem itself; one cannot constrain "X" to be "infinitely close" to, but never exactly, 3. For convenience then, ALPS allows constraints to be written as "$X < 3$" (a strict inequality); this is assumed to mean "$X \leq 3$" (a nonstrict inequality).

That is basically all the necessary information needed to formulate a simple linear program. Creating an LP from a problem scenario is a technique which, with practice, becomes easier. A few other "tricks" are mentioned here; these are useful only to those with some experience in linear programming, but are mentioned as a start:

Certain LP solvers (such as ALPS) allow the use of ***integer variables*** in the formulation. As the name implies, an integer variable can only take on integer values. Although this really creates no additional work in terms of creating an LP, integer programs sometimes require special treatment of the constraints. For example, if "Xa" and "Xb" are integer variables, the constraint

$$Xa + Xb \leq 6.5$$

is somewhat wasteful in terms of the solution process. Integer programs use a different, much lengthier technique for solution than plain linear programs, and as such the programmer should be as efficient as possible (that is, attempt to use as few variables as possible) within the formulation.

In many cases the use of ***binary*** variables is of interest. A binary variable is a special case of the integer variable, one which can only assume the values 0 and 1. Often these variables represent a "go"/"no go" situation in the decision. Also, creating the constraints in a binary program is often tricky. In solving a problem, ALPS treats binary variables the same as integer variables. However, if a problem contains strictly binary variables, ALPS uses a much faster, more efficient technique to solve it.

By combining binary variable with a standard linear program, the user has a large set of new capabilities within the program. First of all, piecewise–linear objective functions can be modeled (functions that are linear with different slopes over several regions). Also, one can include conditional constraints such as "IF X1 + X2 > 2 THEN X3 < 4." These are advanced topics in linear programming; for more information, see a full LP text.

Solution Reports

Regardless of how the LP was formed, once it is solved, the solution data can be of great value. Several new terms come into play once an LP solution is considered; to facilitate this we shall examine the ALPS solution to the mixing process created above:

ALPS: Linear Program Solution

**** Solution # 1 of 1 ****

*** LP SOLVED WITH OPTIMAL VALUE OF 480.0 ***

Variable	Optimal Value	Reduced Costs
Xa	0.0	–40.0
Xb	4.8	

Row	Constraint Name	Slack/Surplus	Shadow Cost
1		1.6	
2		0.0	20.0

*** RANGES IN WHICH CURRENT SOLUTION WILL NOT CHANGE ***

Ranges on Objective Function Coefficients:

Variable	Current Cost	Max Increase	Max Decrease
Xa	40.0	40.0	INFINITY
Xb	100.0	INFINITY	50.0

Ranges on Right–Hand Side (RHS) Values:

Row	Constraint Name	Current RHS	Max Increase	Max Decrease
1		16.0	INFINITY	1.6
2		24.0	2.66667	24.0

Note that ALPS gives the Optimal Value of all the decision variables. These values (Xa = 0 and Xb = 4.8) taken together comprise an ***optimal solution*** to the linear program. In this case, there is only one optimal solution, known as a ***unique optimum***; however, some problems may produce several, or "multiple" optima. The other solutions are known as ***alternate optimal solutions.***

Associated with these optimal values is the overall ***optimal value*** of the objective function. This is just the sum of the objective function coefficients (defined above) times the optimal value of all the decision variables. In the case of a minimization problem, this is the lowest possible value of the objective function within the given constraints; for a maximization it is the highest possible value.

The solution to this particular LP is rather trivial. With a little thought one can see that, since chemical B has a much higher profit margin than chemical A, the manager really has no reason to produce any amount of chemical A. Then, with all production turned to B, the amount of B "Xb" to produce can be maximized until no hours of process 2 remain, so that 24/5, or 4.8, units of 100 gallons should be produced. However, for discussion purposes, we will continue to analyze the solution.

Along with the optimal values, the solution contains the ***reduced costs*** of each decision variable. Understanding these costs requires a bit of knowledge as to how a linear program is solved.

Consider a linear program where there are **n** variables (after the addition of slack and surplus variables, so that all constraint equations are equalities) and **m** constraints. Immediately, m must be less than or equal to n, or else the constraints will overqualify the system of equations. Now, we consider each feasible point of the system described by the

equations. Some simple linear algebra dictates that, of the n variables, exactly m–n of them must be identically zero. These are known as the ***nonbasic*** variables of a solution; the other m variables are known as the ***basic variables*** and, take as a whole comprise the basis of any given solution.

Finding the optimal point(s) of a linear program then reduces to searching through different bases until the best solution is found. This is not done in any haphazard manner, however. Once a feasible solution (described by a basis) is found, a new one is generated by making a certain nonbasic variable (with a zero value) into a basic variable. Of course, this also requires a basic variable to be transformed into a nonbasic variable. (This procedure is known as a ***pivot*** or ***change of basis***.) Associated with this change is a change in the overall objective function value. The choice of which variable "leaves" the basis and which variable "enters" the basis is made so that the objective function is sure to improve. An optimal solution is found when the objective function can no longer be improved by pivoting.

At any given point in the solution process, every nonbasic variable has an associated ***reduced cost*** (also known as an ***opportunity cost***). This relates to the actual numeric change in the objective function if that variable were to enter the basis. Specifically, it is the increase in the objective function value (or decrease, if the reduced cost is negative) for each unit increase of the decision variable. For that reason, all reduced costs in the final solution of a MAX problem will be negative (indicating that no increases are possible) and all will be positive for a minimization problem. Also, since basic variables are already in the basis, any basic variable (a variable with a nonzero optimal value) will have a zero reduced cost.

The values of the reduced costs in the final solution are sometimes useful for interpreting which decision variables are least significant to the problem. For example, when multiple nonbasic variables exist in the final solution of a MAX problem, the one with the least negative reduced cost will have the least affect (per unit variation) on the optimal value. This is part of the broader field of linear programming known as ***sensitivity analysis***.

In the above problem the decision variable "Xa" is nonbasic (since it has zero as an optimal value) and has a reduced cost of –40.0, indicating that for each unit (remember that here, a "unit" is 100 gallons) that "Xa" were to be increased, the optimal objective value would decrease by (that is, the manager would lose profits of) $40.

The solution output also contains information relating to each equation of the constraint set. (ALPS allows the naming of constraints, for easy reference). The slack or surplus of the constraint (if it exists) is given. If the constraint is an equality, it obviously has no slack or surplus. Likewise, if the user has already converted the problem to standard form, ALPS has now way of knowing, and will report no slack or surplus.

An inequality constraint with no slack or surplus is referred to as a ***tight*** or ***binding*** constraint. Often, this means that the constraint is being used to its full capacity, such as the

constraint on Process 2, above. All hours of Process 2 are being used in the current solution. However, Process 1 is slack by 1.6 units, indicating that Process 1 has 1.6 hours of idle time in its cycle.

A very useful quantity in terms of sensitivity is the ***shadow cost*** of each constraint. Basically, this is the "opportunity cost" of the constraint value. This is best explained by example: in the above problem, the shadow cost related to the constraint on Process 2 is 20. This means that for every increase in the right hand side of the constraint, the objective function will increase by $20. Thus, if the manager could allocate just one more hour of Process 2 time per day, the profit would increase by $20. This shadow cost if very useful when many constraints exist; the solution then indicates (via the shadow cost) which constraints are more critical to the solution.

Another term for the shadow cost is the ***dual cost.*** This is the same entity; another realm of linear programming, ***duality theory***, deals with the so–called "dual" of every linear program. In the dual problem, constraints become decision variables, and decision variables become constraints. Thus the "shadow cost" described above is actually the "opportunity cost" or "reduced cost" for each constraint.

It should be noted that the shadow costs and reduced costs are not always meaningful for every LP. Their usefulness depends largely upon how the LP was formulated. For production–type problems, the dual costs can be very helpful. But for other problems (e.g., binary programs and integers programs), the dual solution is basically useless.

One final set of information produced by ALPS is the ***parametric analysis*** of the problem. This analysis studies the changes possible within the formulation. First, notice the ranges (given as "Max Increase" and "Max Decrease") on the objective function coefficients. These numbers denote the range for each coefficient within which the optimal basis will not change. That is, as long as the objective costs stay within these limits, the optimal values of the decision variables will not change. Note that <u>any</u> change in the objective costs <u>will</u>, however, cause a change in the optimal objective function value (since the coefficients have changed, directly affecting the objective value).

Ranges are also given on the right–hand side values of each constraint equation. As long as each right–hand side constant stays within these limits, neither the objective function value nor the optimal value of the decision variables will change. The only quantities that might vary are the values of the slacks and surpluses for the constraints; this is expected since the right hand side values are changing.

The ranges on the objective function coefficients and constraint right–hand sides are quite useful when the LP is used to model a physical system when the parameters are slightly uncertain. As long as the possible ranges for the parameters stays within the prescribed limits, there is no need to re–solve the linear program. However, if <u>any one</u> parameter should change outside the allowed ranges, the final solution is <u>guaranteed</u> not to be the same as before.

In the example above, examine the range on the first (Process 1) constraint. Notice that it may be increased by an infinite amount, without changing the optimal solution. At first, this may seem counter–intuitive; however, remember from the discussion above that Process 1 is not even being used to full capacity (it has a slack). Process 2 is the limiting factor on the solution.

Impossible Problems

Of course, all the above information on solutions is helpful only if the linear program can be solved in the first place. Why would an LP be un–solvable? There are three possibilities; each will be presented here.

Perhaps the most common non–solution is that the LP is ***infeasible***. Informally, this means there is no set of decision variables that satisfies all the constraints at one time. Formally, this means that the feasible set is empty. In any case, this is most likely a formulation error. The LP probably contains too many constraints, or malformed constraints. The only solution is to check the problem. Otherwise, the infeasible LP means that there actually is no solution.

Typically an LP that is ***unbounded*** also represents a formulation error. In this case, one or more of the decision variables would be allowed to take an infinite value, thus causing an objective function value of positive infinity (for a MAX) or negative infinity (for a MIN). Here the LP is underconstrained, and should be checked.

One final dreadful problem may cause an LP to terminate abnormally. If the LP is found to ***cycle*** (or ***diverge***), this means that the iterative method used to solve the linear program has failed. This is determined when the solution takes more than a prescribed number of iterations. Unfortunately, there is no particular way to avoid this problem; the user can only "fiddle" with the various coefficients until the problem can be solved. On the upside, this type of problem has only been known to occur in two or three of the multitudes of types of linear programs, so it is not much to worry about.

Believe it or not, you have just completed a crash course in linear optimization. The guidelines above give a general outline of practically all the terms and concepts that appear in a linear programming text. With that in mind, anyone with any serious interest in formulating and solving LP's should consult such a text for a thorough discussion of linear programming.

Another Production Planning Problem:

The only practical way to improve upon linear programming knowledge is, as in many fields, with practice. Below is an example problem, along with a "walk–through" formulation.

A furniture maker wishes to determine how many tables, chairs, desks, and bookcases he should make in order to optimize his available resources. These products utilize two different types of lumber, and he has on hand 1,500 board–feet of the first type and 1,000 board–feet of the second. He has 800 man–hours available for the total job. Other pertinent information is as follows:

Product Type	Board–Ft of Lumber #1	Board–Ft of Lumber #2	Man–Hours Needed	Unit Profit
Table	5	2	3	$12
Chair	1	3	2	$ 5
Desk	9	4	5	$15
Bookcase	12	1	10	$10

The craftsman also has a sales forecast and backorders which require him to make at least 40 tables, 130 chairs, and 30 desks; as well as no more than 10 bookcases. Determine the optimal production schedule so as to maximize overall profit.

Solution:

This problem is quite similar to the production problem presented earlier, although it is more complex. First, we determine the decision variables necessary. This is almost intuitive:

Xt = Total number of tables to produce
Xc = Total number of chairs to produce
Xd = Total number of desks to produce
Xb = Total number of bookcases to produce

There are three "resources" which must be constrained – board–ft of lumber #1, board–ft of lumber #2, and total man–hrs. Here again we express the total number of board–ft of lumber #1 in terms of the number of each item produced, and then constrain it to be less than (or equal to) the total number of board–ft available:

$$5\,Xt + Xc + 9\,Xd + 12\,Xb \leq 1500$$

The constraints for lumber #2 and man–hrs are similar:

$$2\,Xt + 3\,Xc + 4\,Xd + Xb \leq 1000$$

$$3\,Xt + 2\,Xc + 5\,Xd + 10\,Xb \leq 800$$

There are also constraints on the minimum number of tables, chairs, and desks, as well as a maximum limit on the number of bookcases:

$$Xt \geq 40$$
$$Xc \geq 130$$
$$Xd \geq 30$$
$$Xb \leq 10$$

Finally, all that remains is to create the objective function. Since the furniture maker wishes to maximize profit, we simply use the unit profits of each item as the objective function coefficients:

$$MAX\ 12\ Xt + 5\ Xc + 15\ Xd + 10\ Xb$$

This completes the linear programming formulation. The reader should enter and solve this LP as an exercise, and interpret the results, using the following questions as a guide:

If the furniture maker could buy an additional 10 board–feet of either type of lumber, or 10 man–hours (all for the same cost), which one should he invest in to give the biggest increase in profit (hint: shadow costs)?

Suppose the craftsman discovers that his estimate on his profits for chairs is incorrect. How "wrong" can he be (that is, what range) before he has to re–solve the problem? What if he discovers an additional 100 board–feet of lumber #1?

It should be noted that, even though one cannot build "3.2 chairs," the above problem was formulated as a pure linear (not integer) program. In general, it is wrong to simply "round off" the answer to a linear program to obtain an integer solution. There is no guarantee that the rounded answer is still optimal (or even feasible!) However, in this problem, the order of the decision variables is about 100, which is greater than most of the coefficients in the problem, so that rounding is not too bad of an answer. If an exact answer is required, however, all the decision variables should be declared as "integer" variables.

A Binary Program: The "Stage Coach" Problem

To show the usefulness of a 0–1 integer ("binary") program, the following example is presented:

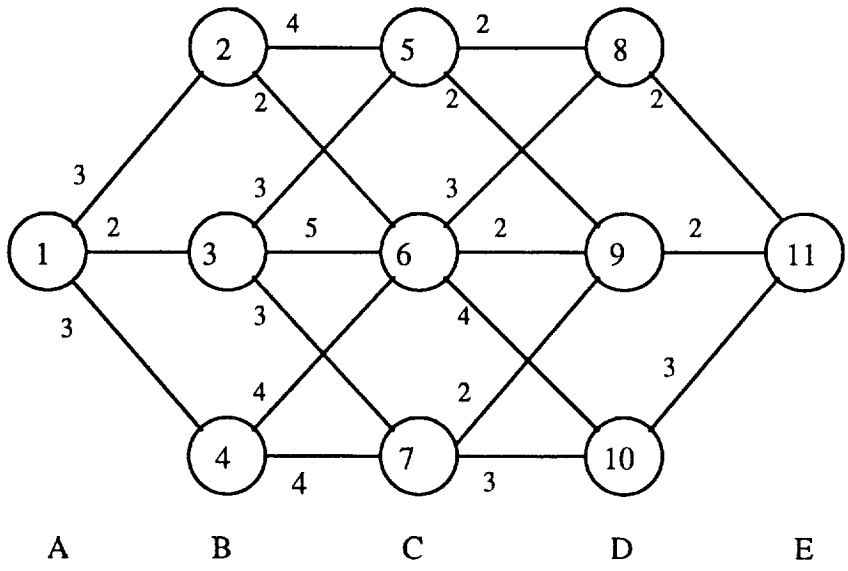

Figure B1 "Map" of Possible Routes Between Five Towns

Figure B1represents a "map" of the possible routes between five towns. Town "A" has one station (number 1), town "B" has three stations (number 2, 3, and 4), and so on. A route is to be found for a stage coach that must travel from town "A" to town "E", with the smallest possible distance. (The number beside each path represents the distance of that path in miles). The stage coach must travel through one station in each town. Determine the optimal (shortest route length) path.

Solution:

The key to this problem is the choice of decision variables. One could choose a single variable "D" as the total distance traveled, create many complex constraints relating to the paths, and then simply minimize "D"; however, this approach would give no information on the actual path to take. A more "observable" approach is to create a binary variable for each path; then let the variable be "1" if the path is taken, and "0" if the path is not taken. Thus we define all the decision variables (the prefix "B" on each variable is an indicator to ALPS the the variable is binary):

BX12 = 1 if the path from station 1 to station 2 is taken, 0 if it is not;

BX13 = 1 if the path from station 1 to station 3 is taken, 0 if it is not;

• • •

BX1011 = 1 if the path from station 10 to station 11 is taken, 0 if it is not.

(There are 20 decision variables in all).

The constraints in this problem are implied; there is no explicit statement pertaining to each decision variable. What we must do is ensure that one and only one route (made up of a set of paths) is chosen. To do this, one can visualize starting from the first station (#1), and moving forward. We note that, from the first station, one and only one path can branch out. Thus we obtain the first constraint:

$$BX12 + BX13 + BX14 = 1$$

In formulating binary programs, it is often helpful to resolve the constraints into simple logical rules, such as "if XX then YY." Once the rules have been defined, it is usually a simple matter to express them in terms of the binary variables. For this stage coach problem, we can use one simple rule to create all the remaining constraints: If a path travels into a station, it must also travel out. The first constraint above ensures that only one path will leave the first node, so there is no chance of more than one route existing at any time. Thus the mathematical expression of the rule above is, for any station n:

$$BXa1n + BXa2n + BXa3n + \ldots = BXb1n + BXb2n + BXb3n + \ldots$$

where a1, a2, a3,... are the station numbers with paths leading into station n and b1, b2, b3,... are the stations numbers with paths leading from station n. For example, for station #2, the rule can be expressed as:

$$BX12 = BX25 + BX26$$

Converting this expression into standard form by moving all variables to the left–hand side gives:

$$BX12 - BX25 - BX26 = 0$$

The entire program formulation follows below. In order to get a better feel for this formulation, refer to the diagram above with each constraint.

BX12	+ BX13	+ BX14				=	1
BX12			– BX25	– BX26		=	0
BX13			– BX35	– BX36	– BX37	=	0
BX14			– BX46	– BX47		=	0
BX25	+ BX35		– BX58	– BX59		=	0
BX26	+ BX36	+ BX46	– BX68	– BX69	– BX610	=	0
BX37	+ BX47		– BX79	– BX710		=	0
BX58	+ BX68		– BX811			=	0
BX59	+ BX69	+ BX79	– BX911			=	0
BX610	+ BX710		– BX1011			=	0
BX811	+ BX911	+ BX1011				=	1

Notice that the last constraint, similar to the first, states that only one path can lead into the last station (#11). Actually, this constraint is not needed — can you explain (in terms of the logic of the constraints) why this is so?

Conclusion

This Appendix has covered the most basic concepts of linear programming, formulating a problem and interpreting the results. It is hoped that the reader will pursue a complete discussion of the merits and use of linear programming, perhaps from the list below. As mentioned before, linear programming is a skill, and can only be mastered with practice. Once mastered, it can be a powerful analytical tool.

Appendix C: Programmer's Guide

1.0. Introduction

This appendix is intended as a guide for those who wish to maintain, expand, port, or otherwise modify the ALPS code. The ALPS code is quite large, and no one (even the original author) could be expected to maintain it without some sort of "roadmap" to the 50+ functions which comprise it. Unfortunately, the author maintains this roadmap mentally; others will be at a distinct disadvantage. However, this guide is intended to make the task of changing ALPS as smooth as possible.

The guide is centered around the six major groups of functions in ALPS. In order of increasing complexity, they are: menus, output routines, formatting functions, file functions, parsing routines, and, the core of ALPS, LP–related functions. Each section of this guide describes each function in various detail: those functions which are quite simple will have little discussion, while those that perform complex tasks will be given a more verbose description. Since the ALPS code contains comments on almost every line, do not look here for explicit references as to which variable is modified, etc. Rather, use these descriptions to get an overall idea what a function does, and how it does it. Then refer to code comments for the details.

ALPS is written in and based upon IBM's APL2/PC product. As such, the programmer will most definitely need a solid APL background. Also, experience in linear programming will be necessary in most cases, although the more superficial functions will not require much LP expertise. However, those attempting to modify any of the simplex method, branch–and–bound, or implicit enumeration code will need experience in these areas. ALPS also relies heavily upon the "AP124" screen Auxiliary Processor; the reader may wish to examine the APL2/PC User's Guide to become familiar with this interface.

2.0. Menu Functions

The menu functions comprise the menu hierarchy of ALPS. In most cases, they simply place the menu on the screen, wait for an input key, and then call the proper submenu or problem–related function, or display an error for an invalid key.

ALPS

This is the top–level function for the ALPS program. Here, the global variables and constants are initialized and the initial screen functions are set up. ALPS calls INTROSC to display the initial menu. From here, any of the submenus FILESC, INTROSC, SOLUTION, and FILEFUNC will be called.

FILEFUNC

All file operations are handled through this menu. Some checks are done before reading a file to make sure the user wants to lose an unwritten problem formulation.

FILESC

This is the function that displays the file options on the screen.

INTROSC

This is the actual introductory menu for ALPS. It displays the menu items onscreen.

SOLUTION

This menu displays options relating to solving the linear program and displaying the solution data. It checks to make sure the problem is actually solved.

SOLVESC

Displays the file options on the screen.

3.0. Output Functions

These functions in some way place output data on the screen (or some other device, in some cases). Typically they are called from the menu functions, and in turn call some of the formatting functions described later.

DEFINESCREENS

This function calls all of the screen–definitions functions for ALPS. All of the screen manipulations in ALPS utilize the "AP124" Auxiliary Processor. See the APL2/PC User's Guide for more information on AP124.

DISPRANGE

This function displays, through the PAGEOUT function, the solution data on optimal ranges for the LP. It uses FORMRANGE to format the output.

DISPSOLUTION

Like DISPRANGE, this function formats the solution data into a text array using FORMRANGE and then displays it to the screen using PAGEOUT.

OVERSC

The "overview" for ALPS is really just a text matrix. This function places the overview text on the screen and allows the user to scroll it around.

PAGEOUT

This function takes any text matrix and displays it to the output screen. It allows the user, via the cursor and paging keys, to scroll around the length and width of the text matrix. It is used by several of the solution output functions.

PRINTREPORT

When the user requests a solution report, this function creates a text matrix of the solution data from FORMSOLUTION and FORMRANGE. It then outputs the text to either a file or a device, using the file function WRITEMATRIX.

SCREEN5

This is actually an input/output screen, which displays a text matrix and inputs a line of text. It is used mainly by the file functions to request a filename from the user.

WRITEOUT

This is an auxiliary function used to output data to a field on the screen. It is actually called by some of the LP–solving functions, to dynamically display data.

DISP_MSG The "message area" is a field on the bottom line of the display screen. This function writes a given string to the message field, and has options to beep and/or pause. It is called by numerous other functions.

GETPROB

This function is actually a self–contained text editor. It is used to allow the user to enter an LP formulation. Several features are included, such as inserting and deleting lines, moving by line or by page, etc. The basic text is stored in a matrix, and most of the function is concerned with display the proper portion of the data, and manipulating the display screen (through the AP124 Auxiliary Processor). When the user ends the edit session, GETPROB also calls the LP input parser to check the problem formulation. If any errors exist, it will display the error message via SCRN11INFO and return to editing.

SCRN11HELP

The input editor GETPROB contains a pop–up help screen, which is actually created and displayed within this function.

SCRN11INFO

This is an auxiliary function to the editor GETPROB. It displays a small pop–up window on the screen, places a text message in the window, and waits for the user to strike a key.

FSBEEP, FSCLOSE, FSCOPY, FSDEF, FSFIELD, FSFORMAT, FSINKEY, FWIWRITE, FSOPEN, FSREAD, FSSCAN, FSSETCURSOR, FSSETFI, FSSHOW, FSUSE, FSWAIT, FSWRITE

These are the "fullscreen" functions from the AP124 workspace. They support all the screen manipulation functions in ALPS. For more information, see the APL2/PC User's Guide.

SCRNDEF1, SCRNDEF11, SCRNDEF2, SCRNDEF4, SCRNDEF5, SCRNDEF6, SCRNDEF8

These are screen definition functions called by DEFINESCREENS. They use the FS–functions described above to create all the various ALPS screen formats.

4.0. Formatting Functions

The functions in this section mainly concerned with formatting matrices (mostly text).

CTRTXT

A function used to center a string of text in a specified width.

DEXB

"Delete extraneous blanks" — that is, remove all occurrences of multiple white spaces.

DLBC

"Delete leading blank columns" — given a matrix of text, this function will remove any empty (white space) columns in front.

DTBR

"Delete trailing blank rows" — this function will strip any empty (white space) rows off a given text matrix.

JUSTIFY, FJUSTIFY

These two functions both fill out a text matrix to a given width (right justify); FJUSTIFY has extra features. Neither is actually called within ALPS, they were just used in development of the help screen text and the overview text.

LJUST, RJUST

Given a string of text, these functions alternately left– and right–justify the text within a field of a given width.

SCRN_EDGE

This function is used to create a white–space edge around a text screen (matrix). Given a matrix size and a text matrix, it will pad the edges.

TOMATRIX

Given any shape of variable for input, this function converts it into a matrix.

UPPER

This function converts a one– or two–dimensional matrix of text into all upper–case characters.

5.0. File Functions

The following functions deal with file I/O in ALPS. DOS files are used to read and write LP formulations, as well as to send the problem solution report to a file (or to a device, which is handled exactly the same as a file). These functions use the APL2 AP210 file Auxiliary Processor.

READMATRIX

Given a valid DOS path/file, this function reads the file and returns a text matrix containing the contents of the file. If there was some error while reading the file, a scalar error code corresponding to an AP210 error will be returned.

WRITEMATRIX

This function writes a DOS file, given the file name and a matrix as arguments. Since storing an entire matrix would be wasteful, this function strips the trailing blanks off each row of the text matrix, and delimits rows with the standard DOS CR/LF sequence. The return will always be a scalar of the status or error of the file write. One important note: the APL/PC User's Guide states that if a file is simply overwritten, it will be replaced. This does not seem to be the case with WRITEMATRIX — it sometimes leaves a garbage file. To ensure a clean rewrite, erase the file with DELETEFILE first.

DELETEFILE

Given a DOS filename, this function will erase the file, permanently. It is used by some of the higher–level menu functions, only when the user has indicated that it is OK to destroy the current file.

ISAFILE

This function simply checks for the existence of a named file. It is used to make sure a file is not overwritten. A return scalar is the AP210 return code; a code of "2" means the file does not exist, while "0" means the file exists.

READPROB

This is a high–level function that reads an LP formulation from disk.

WRITEPROB

This is a high–level function used to store the current formulation to disk. It handles checking to make sure a file is not overwritten, and prompts the user to see if a file should be overwritten.

6.0. Parsing Functions

ALPS contains a very sophisticated text parser that reads an LP formulation and converts it to the data structures necessary to solve the linear program. These are about the second–most complicated functions in ALPS, and require a fair bit of knowledge of linear programming and the revised simplex method to understand.

PARSE

This is the main parsing routine within ALPS. The entire text of the linear program is parsed twice: the first pass collects the names of the variables and constraints, and the second pass actually creates the LP matrices. This function calls the parsing subfunctions below after first combining each "row" of the constraints. After all the parsing is complete, PARSE also determines which (if any) variables are to be marked as integer or binary variables, and set the appropriate data structures. A global variable is defined to denote whether the problem is a pure LP, a mixed integer or integer program, or a pure binary program. Finally, it calls STANDARD to place the linear program in standard form, ready to be solved. If any errors occur while the LP is being parsed, PARSE will create an appropriate error message and return.

PARSENAMES

This function is the "first–pass" of the parser. It is called once for each row in the LP, including the objective function. It simply determines which parts of the row are variable names, and, if the name has not been stored yet, stores the name in the global list. It also checks the end of the line for a comment, and stores this as a "constraint name."

PARSEROW

This is the second pass of the parser. Given a row, it determines which variables are in each row, evaluates the coefficients, and stores them in the global constraint matrix. It also evaluates the constraint sign and right–hand side of the constraint, and stored them as well.

PARSEOF

This routine is used to parse the objective function. Like PARSEROW, it determines the coefficients of the variables in the row and stores these as the objective function costs. STANDARD Once the entire LP has been parsed, STANDARD is called to place the matrix in standard form. This means adding slack and surplus variables, as well as creating two global variables which keep track of which rows contain slack and surplus variables. STANDARD also adds artificial variables to begin the revised simplex method. These artificials are given a high negative cost in the objective function, so as to perform the "big–M" revised simplex method. Also, if the problem is a MIN, STANDARD converts it to a MAX by reversing the sign of all the objective function coefficients.

7.0. LP Functions

The remainder of functions in ALPS deal with the actual solution of the linear program. ALPS solves three different types of linear programs: pure linear programs, integer or mixed integer programs, and binary programs. Each of these problems has its own solution technique: pure linear programs are solved with the revised simplex method; integer or mixed integer programs are solved initially with the revised simplex, and then completed using the branch–and–bound technique; binary programs are solved with the method of implicit enumeration. The revised simplex and branch–and–bound techniques have quite a bit of functions in common, as the solution methods are quite alike. The implicit enumeration routine has separate functions all its own. The type of the LP is determined in the parsing function PARSE, and the solution menu SOLUTION calls the appropriate solving function, SOLVELP, SOLVEIP, or SOLVEBP, as noted below.

One last note – the routines BANDB (for branch–and–bound) and IMPLICIT (for implicit enumeration) are recursive, since they both deal with creating a solution tree. Caveat emptor!

SOLVELP, SOLVEIP, SOLVEBP

One of these routines is called (when the user calls "solve" from the solution menu) for a linear, integer, or binary program, respectively. Each are quite similar, but deal with the specifics of each type of LP. For plain linear programs, SOLVELP calls DOSIMP to perform the simplex method and solve the problem. For integer programs, SOLVEIP calls BANDB to perform the branch–and–bound technique. Finally, for binary programs, SOLVEBP uses IEFORM to place the problem in proper form for solving via implicit enumeration (which is quite different than the standard form for the revised simplex or branch–and–bound techniques), and then uses IMPLICIT to solve the problem. All of the solution routines place all the solution data in the same global variables and format, so that other routines may successfully read the solutions.

DOSIMP

This is the heart of the revised simplex method in ALPS. It is basically one big loop, used to iterate the simplex basis until all reduced costs are negative (remember, all problems at this point have been converted to MAX). This function is passed three matrices describing the simplex problem, and returns the same three matrices in solved form. It uses FINDBV to find the initial basic variables from the constraint matrix. Whenever a pivot operation in the basis is needed, PIVOT is called to compute the new basis. In certain cases, if the initial LP matrix in infeasible, DOSIMP will call DUALSIMP to perform the dual simplex method to describe a feasible problem (this is necessary only during the branch–and–bound procedure, which calls DOSIMP to do the simplex work). Once a solution is found, GETSOLUTION is called to store the solution globally, and GETALTERNATES is called to find any alternate optimal points. FINDBV This is an auxiliary function to DOSIMP. Its job is to

find which variables in the given simplex constraint matrix are basic. For each row in the constraint matrix, there will be exactly one basic variable. FINDBV determines this variable for each row, assuming all the time that the matrix is in standard form.

PIVOT

When the simplex method is to pivot (that is, switch an entering and a leaving variable), PIVOT is called and passed the index of the entering variable. It then globally accesses other simplex variables and determines the leaving variable. From this, it computes the product form for the next basis. A few global variables are updated, and the pivot is complete. PIVOT also takes care of displaying on–screen the current pivot operation, for the status screen in SOLVELP.

DUALSIMP

If DOSIMP has been called by BANDB, it is possible that, after adding constraints to the matrix, the current LP will be infeasible. In this case, DOSIMP will call DUALSIMP to perform the dual simplex method and return the LP to a feasible basis. This is the same type of iteration as used in DOSIMP, but the dual simplex criterion for pivoting is used, and DUALPIVOT is used to perform the pivot operation.

DUALPIVOT

As described above, this routine is used in the dual simplex method to perform pivots. However, unlike PIVOT, DUALPIVOT determines both the entering and leaving variables.

GETALTERNATES

Once the LP has been solved (via DOSIMP), this routine is called to perform any additional pivots to find alternate solutions (from nonbasic variables with zero reduced costs).

GETSOLUTION

If an LP solution has been found, this routine is called to store all the data relevant to the current solution into global variables. This includes calculation of the optimal ranges, shadow costs, slacks and surpluses, etc.

BANDB

This is the main recursive solver for the branch–and–bound technique. Given a current LP, it solves it, and then performs detection of a branching variable. When a branch is performed, the routines ADDUPBOUND and ADDLOBOUND are called to add the bounding constraints on the simplex matrix. BANDB then calls itself to solve the left and right branches of the tree, or exits if the node it is currently evaluating is to be fathomed.

ADDUPBOUND, ADDLOBOUND

These routines are passed an LP formulation in matrix form, and a variable index and bound to add. They simply add a row to the constraint matrix corresponding to the requested

bound. A new LP formulation is returned. Note that this formulation may initially be infeasible, thus requiring the dual simplex method to solve it.

IEFORM

If the problem to be solved is a pure binary program, SOLVEBP will first call this function to convert the problem formulation into the required form for implicit enumeration — that is, the problem must be a max, with all objective function coefficients negative. Variables are substituted as "1 – X" for "X", and a note of which variables were changed is noted for the final solution. The problem is then ready to be solved via implicit enumeration.

IMPLICIT

This is the main recursive implicit enumeration algorithm. A branching variable is chosen using the "least–total–infeasibilities" criterion, by calling DEGINFEAS to determine the degree of infeasibility of each variable. The two branches (fixing the variable at "0" or "1") are created using the FIXVAR routine. Any possible zero completions are stored; otherwise, the branches are evaluated recursively.

DEGINFEAS

Given a variable index, this routine returns the "degree of infeasibility" of the variable in the current implicit formulation. It is used solely to determine a branching variable.

FIXVAR

When IMPLICIT determines a variable to fix, it calls FIXVAR to determine the new implicit formulation. This routine is very memory–conscious — the new formulation actually removes the variable entirely, thus reducing the size of the formulation. This is very useful within the recursive structure of the implicit enumeration solution technique.

ISFEASIBLE

This function determines if the given implicit enumeration formulation is feasible — it is used by IMPLICIT to determine if the node should be fathomed (if the zero completion is feasible) or not.

8.0. Global Variables

Much information about ALPS is contained within the structure of its variables. Certainly the belief that "data structures are the key element to a program" is no more true than in APL2. Examining how these variables are referenced can lead to a lot of insight of the programming of ALPS.

Below is a list of all the key variables within ALPS. Unless otherwise noted, these are global variables. The rank and shape will be noted, where appropriate.

A, b, c

These variables completely determine the original simplex problem. They represent, respectively, the simplex constraint matrix (a matrix), the constraint right–hand sides (a vector), and the original objective function coefficients (a vector). This is the first simplex problem in standard form; that is, all slacks and surpluses have been added.

SLACK, SURPLUS, ARTY

These variables are used to determine which variables represent the slack, surplus, and artificial variables of the LP formulation, since the problem has already been converted to standard form. Each is a vector, their length being the number of rows in the constraint matrix A. The element corresponding to any given row is, for example, in SLACK, the index of the slack variable in that row. If no slack exists in a row, the corresponding element of SLACK will be zero. The format for SURPLUS and ARTY is the same.

PRIMAL, DUAL

These variables are used to store the solution(s), when they are found. They are each matrices, with each row corresponding to a different solution. The first dimension of PRIMAL is the number of variables (since the primal solution represents the values of the variables), while DUAL has the "width" of the number of rows in A, since the dual solution corresponds to rows.

ZVALS

For each row in PRIMAL, there is an element in the vector ZVALS which is the optimal objective function value for the corresponding solution in PRIMAL. This is used within the branch–and–bound and implicit enumeration routines to keep track of all the solutions.

RCOSTS, c_INC, c_DEC, b_INC, b_DEC

These are more variables representing the optimal ranges of any solution. Like PRIMAL, they have a row for each solution.

LP

This is a text matrix, at all times containing the current LP formulation from the editor. In a few cases, the variable LP2 is used as a temporary holding place for this variable.

NAMES, CNAMES

These variables contain the names of the variables and constraints in the LP formulation. They both take the form of a nested array, with each element being a text vector.

DIRTY, CHANGE, LPSOLVED

Each of these variables is a boolean, representing one of the following states: DIRTY is "1" if the current LP formulation has not been saved to disk yet. CHANGE indicates that, usually within the editor, the text of the LP has been changed. LPSOLVED is set to "1" when the linear program currently in LP has been solved, and is reset to "0" whenever the formulation changes.

zero, infinity

These are actually "constants" that are used in the mathematical calculations of the simplex method.

_ZT

This is the internal "zero–tolerance" of ALPS. Any number with a smaller magnitude that _ZT is considered to be zero. Larger problems may have to have this value adjusted.

scrn, helpn

These are nested arrays containing the text of the ALPS display screens and help screen. They are referenced by number within several of the display functions.

bigm

This is a "constant" used in the big–M simplex iterations. All artificial variables are given the value of bigm as an objective function cost.

maxiter

This constant determines how many simplex iterations are carried out before ALPS declares that the given LP is cycling.

References

Gass, S. I., *Linear Programming*, 4th ed., McGraw–Hill, New York, 1975.

Gass, S. I., *An Illustrated Guide to Linear Programming*, McGraw–Hill, New York, 1970.

Ravindran, A., G. Reklaitis, and K. Ragsdell, *Engineering Optimization: Methods and Applications*, John Wiley & Sons, Inc., New York, 1983.

Bradley, S. P., A. C. Hax, and T. L. Magnanti, *Applied Mathematical Programming*, Addison–Wesley, Reading, MA, 1977.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM - 104347		

4. Title and Subtitle	5. Report Date
ALPS—A Linear Program Solver	April 1991
	6. Performing Organization Code

7. Author(s)	8. Performing Organization Report No.
Donald C. Ferencz and Larry A. Viterna	E - 6122
	10. Work Unit No. 474 - 12 - 10

9. Performing Organization Name and Address	11. Contract or Grant No.
National Aeronautics and Space Administration Lewis Research Center Cleveland, Ohio 44135 - 3191	
	13. Type of Report and Period Covered Technical Memorandum

12. Sponsoring Agency Name and Address	14. Sponsoring Agency Code
National Aeronautics and Space Administration Washington, D.C. 20546 - 0001	

15. Supplementary Notes

Donald C. Ferencz, Case Western Reserve University, Cleveland, Ohio 44106 and Summer Student Intern at NASA Lewis Research Center. Larry A. Viterna, Lewis Research Center, (216) 433 - 5398. A copy of the ALPS program (LEW - 14978) is available from Computer Software Management and Information Center (COSMIC), 382 East Broad Street, Athens, Georgia 30602, (404) 542 - 3265.

16. Abstract

ALPS is a computer program which can be used to solve general linear program (optimization) problems. ALPS was designed for those who have minimal linear programming (LP) knowledge and features a menu-driven scheme to guide the user through the process of creating and solving LP formulations. Once created, the problems can be edited and stored in standard DOS ASCII files, to provide portability to various word processors or even other linear programming packages. Unlike many math-oriented LP solvers, ALPS contains an LP "parser" that will read through the LP formulation and report several types of errors to the user. ALPS provides a large amount of solution data which is often useful in problem solving. In addition to pure linear programs, ALPS can solve for integer, mixed integer, and binary type problems. Pure linear programs are solved with the revised simplex method. Integer or mixed integer programs are solved initially with the revised simplex, and then completed using the branch-and-bound technique. Binary programs are solved with the method of implicit enumeration. This manual describes how to use ALPS to create, edit, and solve linear programming problems. Instructions for installing ALPS on a PC compatible computer are included in the appendices along with a general introduction to linear programming. A programmers guide is also included for assistance in modifying and maintaining the program.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Linear programming Optimization	Unclassified - Unlimited Subject Categories 66 and 61

19. Security Classif. (of the report)	20. Security Classif. (of this page)	21. No. of pages	22. Price*
Unclassified	Unclassified	41	A03

NASA FORM 1626 OCT 86

*For sale by the National Technical Information Service, Springfield, Virginia 22161